A Space Island Scene

Arrian Chi William Santosa Ethan Truong Desmond Andersen

***Abstract*—In this simulation, we modeled an island in the middle of space. We started with a simple particle system adapted from the assignments, implemented the Leonard-Jones model to simulate fluid motion, added tentacles to make the environment space-like, and includes a shark. To polish the scene, we smoothed motions with splines and added Japanese decor. In this report, we will describe our process of transforming our simple scene into an oriental space island.**

I. Inspiration

In our initial proposal, we started by brainstorming our implementation process. We wanted to parallelize development while also creating a cohesive scene. To prevent the need to wait for someone else to finish development, we decided to split the project into graphical components by their algorithms. For example, one member can work on the particle system while another works on the tentacles. This way, we could work on the project simultaneously and merge our work together at the end.

With this in mind, we proposed the idea to make a peaceful, Japanese garden scene as it is calming and visually appealing. However, our idea grew into an island as we started to repurpose our particle system to simulate components other than water. Our idea then shifted into a space scene with Japanese elements after learning about inverse kinematics and creating tentacles. Nonetheless, our original goal of making the scene satisfying and peaceful to watch still holds.

In the following sections, we will describe the components implemented, difficulties faced, and end result of the implementation.

II. Waterfall (and Volcano)

A. Fluid Simulation

Fig. 1. Scene

We built off the particle system from the assignment to implement our fluid simulation. In addition to gravity and borders, we added a loop that iterates through all particles and applies a force to each particle based on the Leonard-Jones model. That is, each particle exerts and attractive force and repulsive force on each other particle dependent on the displacement of that particle.

With a few particles (around 100), this implementation works. However, with a large number of particles, the simulation becomes very slow. This was reasonable because the time complexity of the entire operation is O(n$\hat{2}$) (for every particle we are looping for every particle). This subsequently creates a high CPU load and a low frame rate. Below we list some methods of fixing this:

1) Acceleration Data Structure: We may opt to implement an acceleration data structure like an octree, spatial hash grid, or kdtree. The intuition behind these data structures is simple. If we have a particle P and Q in space and Q was located far away from Q, the force exerted by each to each other is trivial to none. Thus, we can partition the space such that for each particle, we only check the particles that contribute significantly to the fluid force. The data structures mentioned do this by keeping track of the partitions particles are located in. A query for the particles within a range then involves a query for the partitions within that range, obtaining the particles we need.
2) Mathematical Intuition: We may move/expand terms around to make the model more efficient to calculate. Above, we see that the masses of particles are involved in the calculation. But if the particles all have a mass of 1 unit, then we don't need to multiply by the mass. We can also expand the unit displacement in the formula and use the original displacement in our calculations. This way, we can avoid one square root operation and one division, which are both expensive operations. Yet another optimization is to avoid calling the divide_by function on the vector and instead multiply_by the inverse of the factor. We avoid doing 2 divisions this way (we do one divide to find the factor instead of three, one for each component of the vector). Finally, in our range calculation, we can avoid a square root operation by comparing the square of the distance to the square of the range. If the square of the distance is less than the square of the range, then the distance is less than the range.
3) Code Optimization: We may improve the code itself. The `Vector` class within `tiny_graphics.js` relies on functions as an analogy for vector operations. Many of these functions call computationally 'slow' methods such as Array.prototype.map(). By inlining the majority

of these functions in performance-critical sections of code, we found signficant reduction in call overhead. Additionally, we cached Vector (`Float32Array`) elements into local variables and object members which provided a slight performance increase in some scenarios. We also unrolled the simulation loop to utilize instruction level parallelism and reduce branch mispredictions (on a low level). We also believe that loop unrolling allows for better register utilization.

4) Hard Limit: The simplest way to reduce the computation is to reduce the number of particles we check statically. So for each particle, we may only check 8 particles, scaling the forces as appropriate. This works because the final force can be throught of as an aggregate of aggregates of forces exerted by particles. A particle that exerts a force onto another will definitely have other particles that exert a similar force to this particle. Thus, magnifying this force by some factor saves us some iterations (at the cost of some accuracy).
5) Symmetric Force: The Leonard-Jones model is symmetric. That is, the force exerted by particle P on Q is the same as the force exerted by Q on P. Thus, we can loop through the particles and apply the force to both particles considered at each iteration. This way, we only need iterate on half the particles.
6) Parallelization: One may also attempt to parallelize the entire particle systems update loop (with multithreading, compute shaders, etc.). This requires 2 synchronization points (barriers), one at the end of the force calculation, and one at the end of the particle position update. This is because the position at the next time step is dependent on the force, which is dependent on the positions of the current time step. If one thread runs faster than another, the timings will be unsynchronized, leading to the slower thread using a future position of a particle to calculate the force, which is incorrect.

In our implementation, we decided that to focus on the animation algorithms, we should implement the optimizations that take the least amount of time, thus opting for the second, third, and fourth optimization algorithms listed above. We attempted to implement the sixth with Javascript webworkers, but the process was too complicated and tedious to implement (JavaScript, unlike C++, does not have a barrier data structure either). Additionally, we also experimented with the theory that the multiple draw calls made is what causes the slow down, but we found that when attempting to combine the particles into one piece of geometry, the simulation becomes even slower (this may be due to the fact that we used the `insert_transformed_copy` into function after clearing out the vertex arrays, making the crux of the workload the initialization of the arrays).

B. Enhanced Particle System

In many game engines, particle systems have controllable parameters that help designers achieve the effects that they want. These may include but are not limited to: particle spawn rate, life time, number of particles, spawn location, spawn radius, exit velocity, particle radius, and so on. These parameters are in addition to the fluid parameters and the gravity parameters required for simulation. We implemented these features after realizing that our waterfall could be repurposed into other systems, such as volcanoes and snow. Some critical components of this system includes:

- Spawn rate: At every timestep, the particle system calculates how many particles to spawn. If all particles spawned at the same time, the particle motion will look too uniform and the stream-like effect is lost. Thus, we spawn particles in chunks per unit time. This is calculated using a gauge mechanism. Assuming that the spawn rate is given per hundredth of a second, the gauge increases by the rate multiplied by the interval. If the gauge is greater than 1, we spawn a particle and decrement the gauge by 1. With this method, we can support spawn rates that are over and under the time step divisions (100 in this case). That is, we can have some timesteps not spawn any particles at all due to slow rates.
- Respawn logic: To implement respawning, we added a lifetime variable to each particle and a check to see if the particle has lived past its lifetime. If it has, rather than deleting it, we zero the lifetime and reset the position and velocity to the source position and exit velocities respectively. That way, we can reuse the same particles over and over again, saving memory and time.

It took a lot of trial and error to get the values we wanted. In the following sections, we outline the methodology of selecting parameters for each component of our simulation.

1) Waterfall: For the waterfall, we know there is a horizontal exit velocity (water trajectory), high attractive force (relative to repulsion), and small spawn radius. Our spawn rate, number of particles, and particle lifespan are set so that the particles can reach the ground, clump together, and despawn shortly. Gravity is also set to reduce the bounce of the particles and repulsion/minimum separation are set so that some particles shoot out randomly (like in stray droplets in real waterfalls). In the end, these parameters create the effect of water gushing out of a cliff and downpouring to the lake below.

2) Volcano: The volcano was actually discovered by accident. When the simulation doesn't have enough particles to spew out (because the particle lifetimes are long and the spawn rate is too high), visually it looks as if the particle system erupted. Combining that with high repulsion, small spawn radius (to let repulsion do its thing), vertical exit velocity, and huge particles, we get a volcano eruption effect. Of course, the volcano loops because of the respawning logic, but this should suffice for now.

3) Snow: Snow is a simple particle system (usually, precipitation is implemented in shaders, but it doesn't matter for the project). Snow covers a huge area, so we must adjust the spawn rate and number of particles to adjust the snow density given the large spawn radius. We set low gravity and

a high repulsion/minimum separation to achieve both the slow fall and small wind. Finally, the lifetime is tuned so that the particles despawn once they hit the ground.

III. Kraken

A. Tentacles

We built the tentacles off of the inverse kinematics assignment, playing around with a variable number of segments, each increasing the degrees of freedom. The tentacle joints are attached to a 3 DOF rotational joint, and our tentacles have 10 segments, thus 30 DOFS. The end effectors are the tips of the tentacles, and the root is the base of the tentacle. Each segment from the root to the tip gets smaller and smaller to create the shape of the tentacle.

The tentacles are animated with inverse kinematics. That is, at every time step, the DOFs of the tentacles are calculated so that the tips move toward the desired position. It involves calculating the displacement (current and desired end effector position) delta, the pseudoinverse Jacobian, and the changes in DOFs (delta thetas). Much of this was outlined in class, and so we will not go over this here. In calculating the Jacobian, we used the cross product between the DOF's axis of rotation and displacement from the joint to the end effector. This formula gives us the instantaneous derivative of the DOF with respect to time. We acknowledge that we compute the derivative numerically by keeping track of the previous positions, believing that the formula gives us the most optimal results.

B. Smoothing

At first, the tentacle animation was very slow, rigid, and glitchy. Thus, we implemented some smoothing algorithms to mitigate these issues.

- Circular tentacle motion: Originally, our implementation had the tentacles move randomly from point to point within a predefined cartesian space. However, we noticed that in both movies and real life, tentacles generally follow a circular fashion. Consequently, we implemented a circular Hermite spline that the tips (end effectors) would follow at each time step. The center of each spline is defined by a point near the tentacles initial position to prevent jittering due to unreachable areas.
- Catmull-Rom transitions: Even after circular motion was implemented, the tentacles still moved rigidly. The problem was that the transition from one circle to the next was discontinuous (after a certain amount of time, the splines are redefined, adding more variance to the animation cycles). To solve this issue, we connected the current position to the starting point of the next circle with a Catmull-Rom "transition" spline. Consequently, the tentacles move smoothly from one circle to the next. The Catmull-Rom spline was used to guarantee smoothness from the previous circle to the transition spline to the next circle. This makes the motion visibly smooth.
- Timing Curve: The motion now was smooth, but the tentacles moved at inconsistent speeds across each curve. So the tentacle would rapidly move for some time, then suddenly slow down. This is because of 2 reasons. First, the delta multiplier for the error in the IK simulation, the epsilon used to determine whether or ot to generate the next desired position, and the $d\theta$ multiplier needed tuning. Second, the tentacle is not guaranteed to move the same amount of distance at every time step. Thus, the desired position should be generated based on a constant arc length. So at every time step, we move the same amount of distance along the current spline. To implement this, we leveraged the arc length table in the spline class, mapping the parametric entry t to the arc length, and the bisection method of searching the parametric entry given the arc length desired. When we need a new desired position, we find the proportion of the arc length with our timing variables, multiply that with the overall arc length of the spline to get the desired arc length, and use the bisection method to find the parametric entry, which we use to find the desired position.
- Finishing Touches: To complete the kraken, we applied a grainy texture over the segments and the joint geometry. We also reduced the number of vertices on the kraken to make the rendering more efficient. There was also a slight pulsation on the tentacles to make them look more alive and alien-like. Finally, the movement was made faster to make the kraken look more aggressive.

IV. Miscellaneous

A few features were added to make the scene more cohesive and visually appealing. Namely:

- Skybox: A night skybox is added to give the scene a spacey, dark vibe. In fact, this space background is what inspired the space island idea. To create the skybox, a night sky texture was taken and then mirrored vertically to create a seamless transition when repeating it side by side. The texture was then mapped onto a subdivision sphere to create the resulting skybox.
- Shark: Mass spring dampers from assignment 1 were used to create shark in the pond.
- Color Spline: We added a color spline to the particles, so that the particles change color according to their squared speed relative to a maximum speed. This is how we made the waterfall have a little white at the bottom, and why the volcano has red particles spewing out but brown rocks on the ground.
- Spline Camera: We created the option for the camera to follow a spline and look towards the center of the scene. The camera follows a C^2 continuous spline path that goes around the island and the lake.
- Decor: We added some decor to the scene to give it a more oriental feel. This includes a torii gate, a bridge, and some lanterns.
- Water: We added a water plane to the scene to give the scene a more realistic feel.
- PBR Material Support: We added basic support for normal maps, ambient occlusion maps, and roughness

maps within an extension of the textured Phong shader to provide more realism to our materials.

V. Conclusion

We have outlined the process of creating our simulation project, starting from the fluid particle system to the inverse kinematics tentacles to the construction of the final scene. Overall, we are happy with our final product and will use the techniques learned in future projects.